Multiplier-Free Implementation of Galois Field Fourier Transform on a FPGA

Sree Balaji Girisankar, Mona Nasseri, *Member, IEEE*, Jennifer Priscilla, Shu Lin, *Life Fellow, IEEE*, and Venkatesh Akella, *Member, IEEE*

***Abstract*—A novel approach to implementing Galois field Fourier transform (GFT) is proposed that completely eliminates the need for any finite field multipliers by transforming the symbols from a vector representation to a power representation. The proposed method is suitable for implementing GFTs of prime and nonprime lengths on modern FPGAs that have a large amount of on-chip distributed embedded memory. For GFT of length 255 that is widely used in many applications, the proposed memory-based implementation exhibits 25% improvement in latency, 27% improvement in throughput, and 56% reduction in power consumption compared to a finite field multiplier-based implementation.**

***Index Terms*—Galois field Fourier transform, RS-codes, BCH-codes, FPGA.**

I. Introduction

Fourier Transform over a Galois (finite) field (GFT) and its inverse (IGFT) are some of the most computationally demanding tasks in the implementation of Bose-Chaudhuri-Hocquenghem (BCH) and Reed-Solomon (RS) codes. Finite field multiplication is the bottleneck in implementing the GFT, so several techniques have been proposed in research literature to reduce the number of multiplications required. A substitution and pre-computation based technique is proposed in [1] to compute GFT for prime lengths greater than 2 over $GF(2^m)$ for arbitrary m that saves about one-quarter of the multiplications compared to a brute-force implementation. There have been research efforts that use a FFT style implementation of GFT to reduce the number of multiplications [2]–[4]. In [5] researchers have proposed methods that can reduce the number of multiplications from n^2 to $\frac{1}{4}n(\log_2 n)^2$ over $GF(p^m)$ where p is a prime value and $n \log_2 n$ for $GF(2^{2r})$, for an arbitrary value of r, and Wu *et al.* [6] improve this further to $O(n(\log_2(n))^{log_2^{3/2}})$ using the cyclotomic method. Ghouwayel *et al.* [2] present the hardware design and implementation of cyclotomic Fast Fourier Transform (CFFT) over $GF(2^m)$ by reformulating the method presented in [3]. Though the architecture has some advantages because some of the results of the computation are reused instead of computing them again, the number of finite field multipliers used in their method is the same as in [5].

In this brief we propose a technique to implement GFT that does not use any finite field multipliers. The main insight of this brief is that by transforming the GFT computation from a *vector* representation to a *power* representation, we can replace multiplication by wrap around carry addition. The challenge is to do the conversion from the vector to power and back efficiently. We proposed to use the extremely large on-chip embedded memory available in modern FPGAs as ROM (read-only memory) to store the precomputed conversion tables. On a Virtex 7 there is about 37 Mb of embedded memory (called Block RAM). This is sufficient to create the ROMs for GFT computation on finite field up to a length of 1023, which meets the requirements of many emerging applications. For example, in the new iterative soft-decision decoding of RS codes [7] - the application that motivates the work presented in this brief - GFT of length 127 is required. Recent FPGAs such as Stratix 10 TX2800 from Altera has about 229Mb of memory with over 11721 M20K blocks. With more memory, the proposed scheme can be easily extended to larger field sizes if necessary. At the end of this brief we also propose a simple technique to reduce the memory requirements for prime length implementation by taking advantage of the cyclic property of multiplications over Galois Field.

In an FPGA the embedded memory blocks can be configured as extremely wide word ROMs. So, it allows a highly parallel vector processing style implementation, which results in extremely low latency and very high throughput. Note that this is only possible because the embedded memory on a FPGA is very flexible and can be configured with extremely large word size. For example, when n is 1023, the proposed architecture accesses 1023 10-bit words (i.e., 10230 bits) in parallel each clock cycle (227 MHz) which represents an aggregate memory bandwidth of 2.3 Tb/s. This allows the computation of GFT of length 1023 in 1027 clock cycles with a throughput of 2.5 Gbps, using about 52% of the on-chip memory resources on the FPGA.

The rest of this brief is organized as follows. We start with a high level overview of the GFT computation and how we propose to implement it by converting it from vector to power representation using ROMs in Section II. Next, in Section III we describe the details of the proposed architecture and a method to reduce the memory requirements in some cases. We present the results of our implementation on a Virtex 7 FPGA

Manuscript received November 1, 2018; accepted December 29, 2018. Date of publication January 21, 2019; date of current version November 1, 2019. This brief was recommended by Associate Editor C. W. Sham. *(Corresponding author: Venkatesh Akella.)*

The authors are with the Department of Electrical and Computer Engineering, University of California at Davis, Davis, CA 95616 USA (e-mail: akella@ucdavis.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSII.2019.2894361

TABLE I
$GF(2^3)$ POWER AND VECTOR REPRESENTATION FOR PRIMITIVE POLYNOMIAL $p(a) = 1 + a + a^3$

Power representation	Vector Representation
-1	(0 0 0)
β^0	(0 0 1)
β^1	(0 1 0)
β^2	(1 0 0)
β^3	(0 1 1)
β^4	(1 1 0)
β^5	(1 1 1)
β^6	(1 0 1)

and comparison with related work, including a traditional finite field multiplier based implementation in Section V.

II. HIGH LEVEL OVERVIEW OF THE PROPOSED SCHEME

$GF(2^m)$ represent the Galois field of 2^m where m is a positive arbitrary integer and $n = 2^m - 1$. Suppose $\boldsymbol{a} = (a_0, a_1, \ldots, a_{n-1})$ is a vector over $GF(2^m)$ and β is a primitive element in $GF(2^m)$ such that $\beta^n = 1$ and every nonzero element α in $GF(2^m)$ can be expressed as β^j,$0 \leq j \leq n-1$. The GFT of $\boldsymbol{a}$ is a vector $\boldsymbol{b} = (b_0, b_1, \ldots, b_{n-1})$ whose t^{th} component, for $0 \leq t \leq n$ is given by,

$$b_t = \sum_{k=0}^{n-1} a_k \beta^{kt} = a_0\beta^0 + a_1\beta^t + a_2\beta^{2t} + \cdots + a_{n-1}\beta^{(n-1)t} \quad (1)$$

This is the inner product of $(a_0, a_1, \ldots, a_{n-1})$ and $(1, \beta^t, \beta^{2t}, \ldots, \beta^{(n-1)t})$ [8]. A symbol in $GF(2^m)$ can either be represented in vector form or its corresponding power form shown in Table I.

Addition can be implemented in hardware very efficiently using the vector representation using just simple XOR gates, but multiplication is challenging because it needs a finite field multiplier. However, the power representation can be used for the multiplication as follows. Equation (1) can be represented in matrix notation:

$$\begin{bmatrix} b_0 & b_1 & b_2 & \ldots & b_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & \ldots & a_{n-1} \end{bmatrix}$$
$$\times \begin{bmatrix} \beta^0 & \beta^0 & \beta^0 & \ldots & \beta^0 = 1 \\ \beta^0 & \beta^1 & \beta^{2\times 1} & \ldots & \beta^{n-1} \\ \beta^0 & \beta^2 & \beta^{2\times 2} & \ldots & \beta^{(n-1)\times 2} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ \beta^0 & \beta^{n-2} & \beta^{2\times(n-2)} & \ldots & \beta^{(n-1)\times(n-2)} \\ \beta^0 & \beta^{n-1} & \beta^{2\times(n-1)} & \ldots & \beta^{(n-1)\times(n-1)} \end{bmatrix} \quad (2)$$

For prime length n, the exponents of β in a row (column) are distinct. Note that the powers of β are computed as $(k \times t)$ mod $(2^m - 1)$. The row vector of inputs $\boldsymbol{a}$ is variable but values inside the square matrix follows the fixed pattern and can be stored in read-only memory (ROM) as a look up table for use in arithmetic operations. For example, consider the GFT for a vector $\boldsymbol{a}$ of length 7 in $GF(2^3)$ where $n = 7$ and $m = 3$. Using Equation (1), the values that are to be computed are,

$$b_0 = a_0\beta^0 + a_1\beta^0 + a_2\beta^0 + a_3\beta^0 + a_4\beta^0 + a_5\beta^0 + a_6\beta^0$$
$$b_1 = a_0\beta^0 + a_1\beta^1 + a_2\beta^2 + a_3\beta^3 + a_4\beta^4 + a_5\beta^5 + a_6\beta^6$$
$$b_2 = a_0\beta^0 + a_1\beta^2 + a_2\beta^4 + a_3\beta^6 + a_4\beta^1 + a_5\beta^3 + a_6\beta^5$$

Algorithm 1 Computing GFT

```
GFT({$a_0 a_1 a_2 \cdots a_{n-1}$})
  Initialize {$b_0 b_1 b_2 \cdots b_{n-1}$} = 0
  for $i = 0$ to $n - 1$ do
    for $k = 0$ to $n - 1$ do
      $x$ = Power($a_k$)
      $y$ = $(x + (i * k) \mod n) \mod n$
      $z$ = Vector($y$)
      if $a_k == 0$ then
        $b_i = b_i \oplus 0$
      else
        $b_i = b_i \oplus z$
      end if
    end for
  end for
```

$$b_3 = a_0\beta^0 + a_1\beta^3 + a_2\beta^6 + a_3\beta^2 + a_4\beta^5 + a_5\beta^1 + a_6\beta^4$$
$$b_4 = a_0\beta^0 + a_1\beta^4 + a_2\beta^1 + a_3\beta^5 + a_4\beta^2 + a_5\beta^6 + a_6\beta^3$$
$$b_5 = a_0\beta^0 + a_1\beta^5 + a_2\beta^3 + a_3\beta^1 + a_4\beta^6 + a_5\beta^4 + a_6\beta^2$$
$$b_6 = a_0\beta^0 + a_1\beta^6 + a_2\beta^5 + a_3\beta^4 + a_4\beta^3 + a_5\beta^2 + a_6\beta^1$$

The computation $a_k\beta^t$ is traditionally done using a GF multiplier where both a_k and β^t are in vector form. In the proposed hardware implementation we convert the input symbol from vector representation to power representation so that both a_k and β^t are in power form. The powers are then added using Wrap around carry-addition. Finally the power representation is converted back to vector representation. For example, consider the computation of $a_k * \beta^4$ where $a_k = 001$ and $\beta^4 = 110$. Instead of standard GF multiplication we will convert a_k into its power representation from the vector representation which give us β^0 according to Table I. So, $a_k * \beta^4$ becomes $\beta^0 * \beta^4$ which is β^4 in the power representation. We convert this back to the vector representation which gives us 110 using Table I.

The method is summarized in Algorithm 1. Where Power(x) and Vector(x) denote power and vector representations of x respectively.

III. PROPOSED ARCHITECTURES

We propose two architectures - Serial In Parallel Out (SIPO) and Parallel In Serial Out Architecture (PISO) which differ in how the inputs arrive and how outputs are produced. In SIPO architecture (see Fig. 1) inputs a_0 to a_{n-1} come in *serially*. PowerROM converts vector representation to power representation. The depth of this ROM is $n + 1$ and the width is $m = \log_2(n + 1)$ bits. So, the size of the PowerROM is $(n + 1) \times \log_2(n + 1)$. BetaROM consists of powers that are needed to be added with the inputs. The size of BetaROM is $n + 1 \times (\log_2(n + 1) \times n)$. In both the architectures, the values of BetaROM are pre-computed. The adder unit performs wrap-around carry addition where the carry is added back to get the final result. Then the VectorROMs are used to convert from power to vector representation. The size of each VectorROM is $(n + 1) \times log_2(n + 1)$. If dual-port memory is used the number of VectorROMs needed is $\lfloor \frac{n}{2} \rfloor$. The output

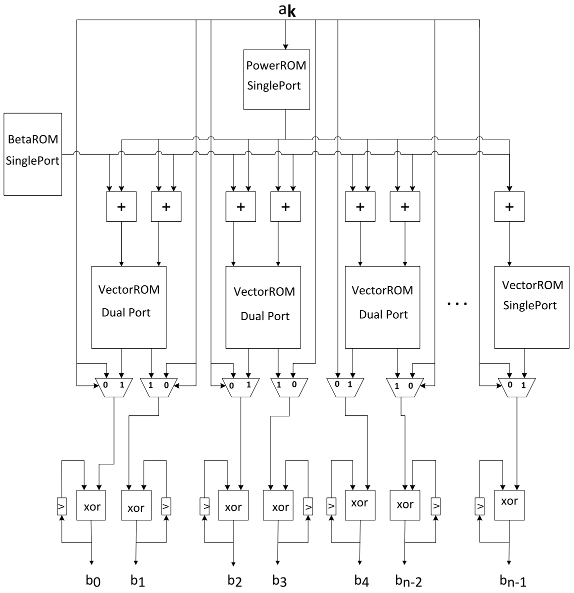

Fig. 1. Serial In Parallel Out(SIPO) Architecture.

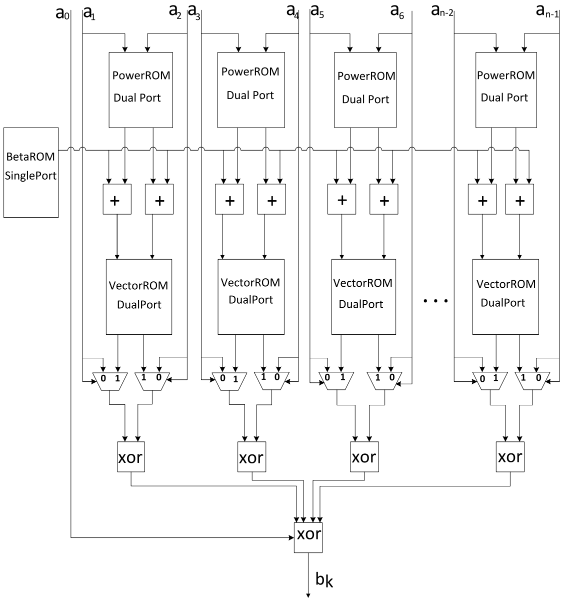

Fig. 2. Parallel In Serial Out(PISO) Architecture.

of the VectorROM goes to the multiplexers. The Multiplexer selects outputs zero if the input is zero or selects the output of VectorROM otherwise. These are accumulated together by Galois field addition, i.e., using XOR gates. In this architecture once a_k is received, $a_k * \beta^z$ is calculated in parallel for all z values according to k^{th} column of the square matrix in equation (2). After all the inputs have arrived, the outputs b_0 to b_{n-1} are available simultaneously.

In PISO architecture shown in Fig. 2, inputs a_0 to a_{n-1} are assumed to be available in parallel, so we start computing the outputs from b_0 to b_{n-1} one by one (serially). PowerROM converts vector representation to power representation. The depth of this ROM is $n+1$ and the width is $\log_2(n+1)$ bits. The main difference between SIPO and PISO is that we replicate multiple copies of this PowerROM to facilitate parallel lookup. With dual-port ROMs, the number of PowerROMs needed is $\lfloor \frac{n}{2} \rfloor$. BetaROM consists of powers that are needed to be added with the inputs. The size of BetaROM is $(n+1) \times (log_2(n+1) \times n)$. The adder unit is wrap-around carry addition where the carry is again added with the result to get the final result. Then the VectorROMs are used to convert from power to vector representation. The size of each VectorROM is $n + 1 \times \log_2(n+1)$. The number of VectorROMs needed are $\lfloor \frac{n}{2} \rfloor$. These outputs are XOR-ed together (as in the previous case) to get the final output. The control unit for this architecture is very simple as it just generates the address for the BetaROM.

TABLE II
THE POWER REPRESENTATION OF $\beta^j * \beta^z = \beta^{j+z}$

z \ j	0	1	...	n-1
0	β^0	β	...	β^{n-1}
1	β	β^2	...	β^0
2	β^2	β^3	...	β
...	...	...	...	...
$n-1$	β^{n-1}	$\beta^n = 1$	...	β^{n-2}

TABLE III
EXAMPLE OF POWER REPRESENTATION OF β^{j+z} FOR $GF(2^3)$ AND PRIMITIVE POLYNOMIAL $p(a) = 1 + a + a^3$ (WHERE $\beta^0 = \beta^7 = 1$)

z \ j	0	1	2	3	4	5	6
0	1	β	β^2	β^3	β^4	β^5	β^6
1	β	β^2	β^3	β^4	β^5	β^6	1
2	β^2	β^3	β^4	β^5	β^6	1	β
3	β^3	β^4	β^5	β^6	1	β	β^2
4	β^4	β^5	β^6	1	β	β^2	β^3
5	β^5	β^6	1	β	β^2	β^3	β^4
6	β^6	1	β	β^2	β^3	β^4	β^5

IV. ARCHITECTURE OPTIMIZATION FOR PRIME LENGTH

For prime length GFT computations, the cyclic property of Galois field can be used to decrease the memory usage of proposed architectures. The idea here is to form a table of the multiplication results of $a_k * \beta^{kt}$ in (1) over $GF(2^m)$ which are distinct values. As mentioned before a_k an element in $GF(2^m)$ can be expressed as β^j. For simplicity it is assumed $\beta^{kt} = \beta^z$. Table II shows the power representation of $\beta^j * \beta^z = \beta^{j+z}$ for $0 \leq z, j < n$ and Table III shows the β^{j+z} for $GF(2^3)$ and primitive polynomial $p(a) = 1 + a + a^3$. When β^j is zero, the result is zero too. Note that each column of the Table II for $1 \leq j < n$ can be obtained by cyclic shift of the $j = 0$ column by j. Each column in Table II and Table III includes distinct values. This property is true when n is prime.

Table IV shows the vector representations of the β^{j+z}s of Table III. The content of each column is the shifted version of the second column (that belongs to $\beta^0 = 1$), by j. For example for input equal to 5, the second column is shifted by 6, because the power representation of 5 is β^6. The second column of Table IV is the vector representation of β^0 to β^6 which is available to us in the VectorROM. As a result, in the optimized method, BetaROM and PowerROMs can be removed from the implementation shown in Fig. 1 and replaced by a smaller ROM of size n which contains the indices of the

TABLE IV
VECTOR REPRESENTATIONS OF THE β^{j+z}S FOR $GF(2^3)$ AND PRIMITIVE POLYNOMIAL $p(a) = 1 + a + a^3$

z \ β^j	1	2	4	3	6	7	5
0	1	2	4	3	6	7	5
1	2	4	3	6	7	5	1
2	4	3	6	7	5	1	2
3	3	6	7	5	1	2	4
4	6	7	5	1	2	4	3
5	7	5	1	2	4	3	6
6	5	1	2	4	3	6	7

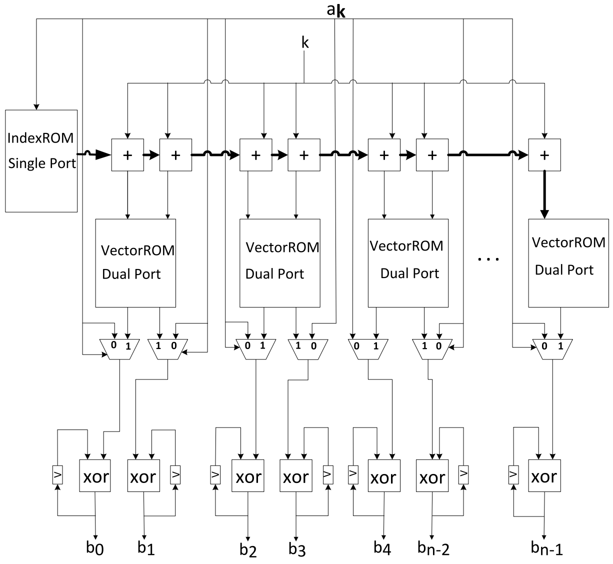

Fig. 3. Optimized prime length SIPO architecture with critical path highlighted consisting of $n - 1$ full adders.

VectorROM. We call this the IndexROM. Therefore to calculate $a_k * \beta^{k\times j}$, we look up the index value of the input in the IndexROM, which is simply the a_k^{th} element of IndexROM, then to generate the $a_k * \beta^{k\times j}$s for $1 \leq j < n$, just add the index to $k \times j$ and then look up the VectorROM to obtain the result. Fig. 3 shows the optimized architecture for $GF(2^3)$ and example below illustrates the details of the optimization.

Assume input $a_k = 2$ and $k = 3$. According to Equation (1), a_3 should be multiplied by: $[\beta^0, \beta^3, \beta^6, \beta^2, \beta^5, \beta^1, \beta^4]$. We generate IndexROM that contains indices of the VectorROM. The contents of the VectorROM and IndexROM are as follows. VectorROM=[1, 2, 4, 3, 6, 7, 5], IndexROM=[1, 2, 4, 3, 7, 5, 6]. Since $a_3 = 2$, its index in IndexROM is 2. The vector representation of $a_3 * \beta^0$ is the second element of VectorROM (because VectorROM[2]=2). To compute the rest of the values, the shifted value which is $k \times j$ should be added to 2 (the index). For $a_3 * \beta^3$, one can add the index to k ($k = 3$) which results in 5, therefore VectorROM [5]=$a_3 * \beta^3 = 6$. Similarly to calculate $a_3 * \beta^6$, index is added to $2 \times k$; $2 + 2 \times k = 1 \ mod\ 7$, and VectorROM[1]=1, then $a_3 * \beta^6 = 1$, and so on.

V. EVALUATION AND RESULTS

The proposed SIPO architecture was synthesized and implemented on a Xilinx xc7vx485tffg1761-2 Virtex 7 FPGA using Vivado version 2017.4 tools. Resource utilization, timing, throughput and power consumption of the architecture for various values of n and $GF(2^m)$ are tabulated in Table V. For $n \leq 63$, number of BRAMs used is 0 because synthesis tools do not infer BRAMs for smaller memories. As n increases, throughput increases due to the higher parallelism in the proposed architecture and the peak throughput is 3.64 Gbps and then the throughput decreases. Note that this is because of the increase in interconnect delay as the fanout of the PowerROM increases from 255 to 1023 which results in a decrease in the clock frequency from 547 MHz to 227 MHz. Note that the clock frequency is high because the design uses very few logic resources (about 11% even for the largest design) as most of the logic is simple XOR gates and full adders which can be implemented very efficiently on a modern FPGA. On a Virtex 7 the proposed architecture scales up to $GF(2^{10})$ beyond that the available Block RAM becomes the bottleneck, but as we discussed above, the optimized architecture can be used to reduce the ROM requirements which would allow GFT of length greater than 1023. However, length $n = 1023$ or less is adequate in most applications.

Table VI shows the results of GFT implementation on the exactly the same FPGA (Xilinx xc7vx485tffg1761-2) using finite field multipliers for comparison. We use hybrid Karatsuba multiplier as in [9] with Montgomery reduction array for the implementation of GFTs. This is the most efficient multiplier design for the size of multipliers required for implementing the GFTs of interest in this brief. We use Gappmair algorithm [1] for prime length GFTs, i.e., when n is 7, 31, 127, and Good-Thomas algorithm [10] when n is not prime, i.e., for 15, 63, 255, 511, and 1023. Good-Thomas algorithm is efficient because it uses a divide and conquer strategy but it can only be used if the factors are prime. Given the difference in algorithms used, the results in Table VI can be counter-intuitive in some cases - for example, the resource utilization for n =255 is lower than n =127 and the power consumption of n =31 is higher than the implementation for n =63. Overall the results show that memory based implementation not only requires fewer FPGA resources such as LUTs and slice registers but also exhibits lower latency, higher throughput and significantly lower power consumption. For GFT of length 255 that is widely used in many applications, the proposed memory based implementation exhibits 25% improvement in latency, 27% improvement in throughput, and 56% reduction in power consumption compared to a finite field multiplier based implementation. In terms of resource utilization, the proposed implementation requires significantly less slice registers (4399 vs 17283) and LUTs (7106 vs 11885) but uses 94 Block RAMs, which a multiplier based implementation does not need.

Figure 4 shows that the proposed memory based implementation is better than a multiplier based implementation as the size of the GFT increases. This is because a multiplier based implementation uses almost 13.5X more slice registers and 4.5X more lookup tables when we go from $n = 511$ to $n = 1023$ which more than offsets the power consumption of the clocked memory blocks. Furthermore, given the utilization of the FPGA is more in the multiplier based implementation, the interconnect power is significant also. In an

TABLE V
RESOURCE UTILIZATION AND PERFORMANCE OF THE PROPOSED MULTIPLIER-FREE GFT IMPLEMENTATIONS

	n=7 GF(2^3)	n=15 GF(2^4)	n=31 GF(2^5)	n=63 GF(2^6)	n=127 GF(2^7)	n=255 GF(2^8)	n= 511 GF(2^9)	n=1023 GF(2^{10})
Slice Registers	113	283	685	1616	2871	4399	9804	21767
Slice LUTs	103	274	699	1785	4912	7106	15643	35678
Block RAMs	0	0	0	0	13	94	193	542
Max Freq (MHz)	666.667	666.667	645.161	547.945	524.934	333.333	275.862	227.273
Latency (Clock Cycles)	7+4	15+4	31+5	63+3	127+4	255+4	511+4	1023+4
Throughput (Gbps)	1.98	2.64	3.2	3.2	3.64	2.64	2.43	2.2
Power (W)	0.271	0.280	0.373	0.501	1.303	1.516	3.329	6.282

TABLE VI
RESOURCE UTILIZATION AND PERFORMANCE OF MULTITPLIER-BASED GFT IMPLEMENTATIONS

	n=7 GF(2^3)	n=15 GF(2^4)	n=31 GF(2^5)	n=63 GF(2^6)	n=127 GF(2^7)	n=255 GF(2^8)	n= 511 GF(2^9)	n=1023 GF(2^{10})
Slice Registers	145	256	3401	1254	42115	17283	104783	295005
Slice LUTs	98	100	3145	768	37540	11885	96405	161409
Block RAMs	0	0	0	0	0	0	0	0
Max Freq (MHz)	412.03	367.78	377.5	311.91	324.2	268.88	238.52	207.12
Latency (Clock Cycles)	7+2	15+3	31+3	63+3	127+3	255+4	511+4	1023+4
Throughput (Gbps)	1.23	1.44	1.85	1.86	2.1	2.08	2.07	2.1
Power (W)	0.293	0.317	0.716	0.476	4.47	3.44	5.71	8.67

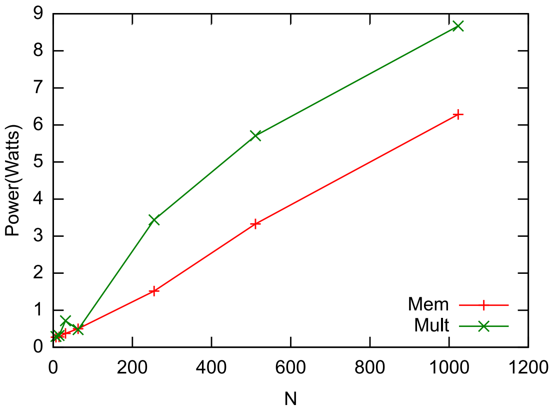

Fig. 4. Scaling Power with GFT Size.

ASIC implementation these trade-offs might be different - for example, the multiplier based implementation will be realized directly with complex gates instead of look-up tables and the interconnect can be more efficient as well (does not have to go through repeaters), so the multiplier based implementation might be more competitive with the proposed memory based implementation when it comes to power consumption.

VI. CONCLUSION

Galois Field Fourier Transform (GFT) is an important operation in signal processing and digital communication applications that rely on RS and BCH codes. Though there have been many advances in reducing the number of finite field multipliers required to compute the GFT, it is still a bottleneck especially when it comes to n FPGA implementation. In this brief we show how on-chip embedded memory can be used to implement GFTs without using any multipliers. We show that the proposed designs are superior to multiplier based implementation in terms of latency, throughput and power consumption in addition to register and LUT requirements. The Inverse Galois Fourier transform (IGFT) of $\boldsymbol{a}$ is a vector $\boldsymbol{b} = (b_0, b_1, \ldots, b_{n-1})$ whose t^{th} component, for $0 \leq t \leq n$ is given by, $b_t = \sum_{k=0}^{n-1} a_k \beta^{-kt} = a_0\beta^0 + a_1\beta^{-t} + a_2\beta^{-2t} + \cdots + a_{n-1}\beta^{-(n-1)t}$, so the same architecture and implementation scheme can be used for IGFT as well.

REFERENCES

[1] W. Gappmair, "An efficient prime-length DFT algorithm over finite fields GF(2^m)," *Eur. Trans. Telecommun.*, vol. 14, no. 2, pp. 171–176, 2003.

[2] A. Al Ghouwayel, Y. Louët, A. Nafkha, and J. Palicot, "On the FPGA implementation of the Fourier transform over finite fields GF(2^m)," in *Proc. Int. Symp. Commun. Inf. Technol. (ISCIT)*, Sydney, NSW, Australia, 2007, pp. 146–151.

[3] P. V. Trifonov and S. V. Fedorenko, "A method for fast computation of the Fourier transform over a finite field," *Problems Inf. Transm.*, vol. 39, no. 3, pp. 231–238, 2003.

[4] X. Wu, Z. Yan, N. Chen, and M. Wagh, "Prime factor cyclotomic Fourier transforms with reduced complexity over finite fields," in *Proc. IEEE Workshop Signal Process. Syst. (SIPS)*, 2010, pp. 450–455.

[5] Y. Wang and X. Zhu, "A fast algorithm for the Fourier transform over finite fields and its VLSI implementation," *IEEE J. Sel. Areas Commun.*, vol. 6, no. 3, pp. 572–577, Apr. 1988.

[6] X. Wu, Y. Wang, and Z. Yan, "On algorithms and complexities of cyclotomic fast Fourier transforms over arbitrary finite fields," *IEEE Trans. Signal Process.*, vol. 60, no. 3, pp. 1149–1158, Mar. 2012.

[7] S. Lin, K. Abdel-Ghaffar, J. Li, and K. Liu, "Iterative soft-decision decoding of reed-solomon codes of prime lengths," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, 2017, pp. 341–345.

[8] A. Reyhani-Masoleh and M. A. Hasan, "Low complexity bit parallel architectures for polynomial basis multiplication over GF(2^m)," *IEEE Trans. Comput.*, vol. 53, no. 8, pp. 945–959, Aug. 2004.

[9] C. Grabbe, M. Bednara, J. Teich, J. von zur Gathen, and J. Shokrollahi, "FPGA designs of parallel high performance GF(2^{233}) multipliers," in *Proc. ISCAS*, 2003, pp. 268–271.

[10] I. J. Good, "The interaction algorithm and practical Fourier analysis," *J. Roy. Stat. Soc. B*, vol. 20, no. 2, pp. 361–372, 1958.